A Domain Specific Language Approach for Agent-Based Social Network Modeling

Enrico Franchi
Dipartimento di Ingegneria dell’Informazione
Università degli Studi di Parma
Parma, Italy
Email: efranchi@ce.unipr.it

Abstract—Although in the past twenty years agent-based modeling has been widely adopted as a research tool in the fields of social and political sciences, there is lack of software instruments specifically created for social network simulations. Restricting the field of interest to social network models and simulations instead of supporting general agent-based ones, allows for the creation of easier to use, more focused tools.

In this work, we propose PyNetSYM, an agent-based modeling framework designed to be friendly to programmers and non-programmers alike. PyNetSYM provides a domain-specific language to specify social network simulations expressed as agent-based models. PyNetSYM was created to deal with large simulations and to work effortlessly with other social network analysis toolkits.

I. INTRODUCTION

In the last ten years, the pervasive adoption of social networking sites has deeply changed the web and such sites became an unprecedented social phenomenon [1]. According to [2], web sites have attracted users with very weak interest in technology, including people who, before the social networking revolution, were not even regular users of either popular Internet services or computers in general.

In the preceding decade, Agent-Based Modeling (ABM) was widely adopted as a research tool in the fields of social and political sciences, because a multi-agent system is a suitable means for modeling and performing simulations on complex systems: a model consists of a set of agents whose execution emulates the behavior of the various individuals that constitute the system [3]. The use of multi-agent systems is especially appropriate for the modeling of systems characterized by (i) a high degree of localization and distribution and (ii) dominated by discrete decisions. In particular, the use of such an approach gave important results in social science because it allows a very useful means for studying social phenomena by providing a natural way of representing and simulating not only the behavior of individuals or groups of individuals but also their interactions that concur to the global behavior [4], [5], [6], [7].

In parallel with the theoretical development of ABM a number of software platforms were developed to ease the task of running the simulations; among these the most popular are Swarm [8], Mason [9], RePast [10] and NetLogo [11]. With the exception of NetLogo, the others are rather general purpose, complex software systems designed to handle any kind of agent-based simulation. However, they have many features that significantly complicate the software and that are not particularly useful for social network processes. On the other hand, NetLogo somewhat lacks features that may be useful even for simulations (e.g., some language constructs, modularity, multi-file editing) [12] and most of its features revolve around the idea of spatially embodied agents placed on a bi-dimensional grid.

In this paper, instead, we introduce a different kind of ABM tool specifically created for network generation and general processes over networks. The tool we propose does not aim at being a general purpose agent-based modeling tool, thus overall remaining a simpler software system, whereas it is extensible where it really matters (e.g., supporting different representations for networks, from simple memory-based ones to pure disk-based storage of the network representation for huge simulations).

Simulation parameters can be easily set both programmatically or as command line arguments. This also allows to easily run simulations with different starting parameters, for example to perform sensitivity analysis and any kind of batch execution.

People with little programming skills shall be able to specify and run their simulations, but skilled programmers must not feel limited. Consequently, we designed an agent-based Domain-Specific Language (DSL) created to express and execute (i) social network generation models and (ii) general processes over social networks. The system focuses on expressing communicative acts among the nodes in the network, as they can be used to express any process conveniently (e.g., link creation/removal or information/disease diffusion).

In Section II we expose the motivations for this work more thoroughly and we compare it with the ABM tools mentioned before. In Section III we describe the logical meta-model of the simulation toolkit, and in Section IV and Section V we present the DSL design and the concurrency model, respectively. Eventually, in Section VI we give some experimental results on our system and draw some conclusions.

II. MOTIVATIONS OF THE WORK

At the beginning of this Section we briefly explain what ABM is and, specifically, the general structure of an agent-based simulation; then, in Subsection II-B, we put ABM in the perspective of social network modeling, i.e., we present
the peculiar traits of social network simulations and how they impact on designing an ABM of a social network processes. Eventually, we discuss the state of the art of ABM and how it is related to our project.

A. Agent-Based Simulation Fundamentals

According to [13] an agent-based simulation has three main components: (a) a set of agents, (b) a set of agent relationships and (c) the agents’ environment.

Although the actual definition of agent is still debated ([14], [15], [16]), there is substantial agreement: (i) that agents are autonomous, i.e., they operate without human intervention; (ii) that agents are reactive, i.e., they can react to external events; (iii) that agents are pro-active or have goal-driven behavior; and (iv) that agents provide a defined interface to an arbitrary system. However, the fact that agents in simulations actually have these properties, and especially true autonomy and pro-activeness, is debated in [17]; for a thorough review of agents in the context of social network, we refer to [18].

In agent simulations of social networks, a terminological ambiguity arises between the agent relationships mentioned in (b), that we call system-level relationships henceforth, and the actual relationships that are in the social network (network-level relationships or links). Although agents that are linked in the network do also have a system level relationship (i.e., being linked in the network), the opposite does not hold.

The environment is an important element in agent-based simulations, considering that it provides not only the conditions under which an agent exists, but it also provides, as communication environment, principles, processes and structures that enable information exchange among agents. When agents interact in a coordinate manner, it becomes a social environment [19].

Thus, from the agent-oriented perspective, the true environment for any social network simulation is a representation of the network itself, because agents exist only as nodes in the network; moreover, for most processes the network is also the vehicle for information spreading and, eventually, the social environment where agents meet.

Essentially, while for agent-based simulations there are many different environment topologies (e.g., 2-dimensional grid, 2D/3D euclidean space, network topology, Geographic Information System (GIS)), the network topology is the primary environment topology for social network simulations.

Although, in principle, a network simulation may require a different environment topology, it is rare that there is need for a specific entity explicitly representing the environment, as it is sufficient to add notion of placement inside the agents, that interact, as usual, with communicative acts among themselves.

Eventually, even though proper agent-based models should allow only “local” (according to the environment, i.e., neighborhood relation according to the topology), for many interesting simulations it is required to allow direct communication among non-neighboring agents. Simulation involving weak links [20] are a typical example, in that weak link formation is not easily simulated with only local interactions.

B. Towards Agent-Based Modeling of Social Networks

In the present Subsection we introduce the design principles of PyNetSYM [21], a social network simulation system we propose that has the following defining goals: (i) it must support both small and huge networks; (ii) simulations shall be effortlessly run on remote machines; (iii) it must be easy to use, even for people without a strong programming background; (iv) it must not be significantly harder to deploy a huge simulation than a toy example of few hundred nodes.

A PyNetSYM simulation has four main components: (i) the user interface, (ii) the simulation engine, (iii) the simulation model and (iv) the network database. The simulation model needs to be specified for each simulation and is the only part that has to be written by the user. The user interface is responsible to take input from the user, e.g., simulation parameters or information on the analysis to perform. The simulation engine defines the concurrency model of the simulation, the scheduling strategies of the agents and the communication model among the agents. The network database actually holds a representation of the social network; it may be in-memory or on persistent storage.

In order to support both small (few hundred nodes) and huge simulations (millions of nodes), we need a pluggable network database and specifically we offer a choice between: (i) simple in-memory network database, that is fast and easy to use for small simulations, and (ii) disk based network database for larger networks. In fact, we provide multiple in-memory network databases because different implementations may provide better performance for specific processes and analysis. For example, the task of extracting a random edge from the network is much more efficient if the underlying representation allows direct indexing of edges, as does igraph [22].

Multiple network databases are supported by providing graph wrappers that allow the agents to transparently use the underlying network database. Choosing the actual wrapper is actually matter of a single option when defining the simulation. Adding a support for a new database is as easy as defining a new wrapper; however, the necessity should seldom occur as we plan to provide a large selection of built-in storage solutions. Scaling from small simulation to larger ones is only a matter of choosing the right wrapper.

Considering that we want to cope with large scale simulations, such simulations may be more conveniently run on server class machines with more RAM and better CPUs than those usually found on laptops or desktop machines. So, in order to simplify remote execution, we require an entire simulation to be definable in a single file; however, should the need arise, it is still possible to apply full software engineering techniques and split the model in a nested hierarchy of source files. Single file models can be deployed simply by copying the source program on a remote machine (provided that the rest of PyNetSYM is already installed); moreover, models may also be sent via email or put under version control systems.

A mainly GUI based simulation toolkit may be significantly more complex to use on remote machines; consequently the
main user interface is command-line-based and parameters for the simulation are passed as command line arguments. The design decision is both compatible with the remote execution goal and also allows for batch executions or multiple executions with different parameters. We also support Read-Eval-Print Loop (REPL) interaction in order to dynamically interact with the simulation, e.g., to interactively analyze the network.

In order allow people without a strong programming background to easily write simulations, we decided to create a Domain-Specific Language (DSL). A DSL is a language providing syntactic constructs over a semantic model tailored towards a specific domain (e.g., building software, expressing object-relational mappings). The idea is that DSLs offer significant gains in productivity, because they allow the developers to write code that looks more natural with respect to the problem at hand than the code written in general purpose language with a suitable library. Moreover, a well thought DSL may be more easily understandable for and perhaps written by domain experts. On the other hand, the main perceived drawback of DSLs is the cost of realization, that may be significantly higher than an API based solution, especially considering the additional skills the language developers may need to have with respect to average programmers.

DSLs are usually categorized in internal and external DSLs: an external DSL is a new language that is compiled or interpreted (e.g., makefile), an internal DSL is represented within the syntax of a general purpose programming language, essentially using the compiler/interpreter and runtime of the host language. A more thorough discussion of the subject can be found in [23].

Traditionally, DSLs were sparingly used to solve engineering problems, with the possible exception of XML based DSLs, which are, nonetheless, cumbersome to write and read even if relatively easy to build. The most famous example of XML DSL is probably Ant. Nowadays, with the widespread diffusion of very expressive languages such as Ruby, Haskell or Groovy the general attitude towards DSLs has changed ([24], [25], [26], [27], [28], [29]), especially because writing internal DSLs in such languages significantly lowers the developing costs to a point that is comparable to an API based solution [30].

Consequently, we decided to develop an internal DSL over a very high level programming language. This should provide an environment that is friendly enough for scientists without strong programming background, without being limiting for the others. Moreover, it greatly simplifies external library usage, since all the libraries available for the host language are available for the DSL as well. The actual description of our DSL is detailed in Section IV.

C. State of the Art of ABM

In this Subsection, we review the most widespread ABM toolkits in more detail and compare our design decisions with theirs, where appropriate.

NetLogo focuses on the 2-dimensional grid topology, which is useful for many simulations, but not particularly relevant in our domain. While it is possible to simulate a different topology, many of the powerful features of the toolkit are created with the 2D grid topology in mind.

Another problem we have found with NetLogo is that the visualization of models tends to leak in the rest of the model. Moreover, the visualization features are closely related to the idea of moving turtles on the grid, which feels awkward in case of social networks. For example in the demonstrative models about social networks provided with NetLogo, node-turtles “walk” on the grid while the simulation progresses just to present a visualization of the social network. On the other hand, social network analysis packages (e.g., NetworkX [31]) offer easier to use, more efficient and more scalable alternatives.

Eventually, although we agree that agent-based simulations benefit from a custom language, we feel that the NetLogo language lacks: (i) many interesting features found in general purpose languages that may be useful to implement algorithms and (ii) many useful libraries (e.g., proper social network analysis toolkits, statistical toolkits). In fact, using network analytic or statistical techniques is hardly uncommon inside a model, and many of these algorithms may well be as complicated to implement as the simulation itself, especially considering that NetLogo is also aimed at non-programmers.

In fact, out approach is more similar to that of ReLogo which is a Groovy DSL inspired by NetLogo and built upon the RePast Symphony framework. Since ReLogo models are written in Groovy, they are able to call any Java library providing the required functionality. However, ReLogo is still designed with the 2D grid in mind and it does not easily interoperate with the rest of the RePast framework [32].

Mason, RePast and Swarm follow an approach different from our own. They are mostly libraries or frameworks for general purpose languages (Java for Mason and RePast and Objective-C or Java for Swarm) and this is a clear advantage when interfacing with external libraries (e.g., the statistics or network analytic packages we already mentioned), to the point that RePast, for example, already provides integration with Jung. However, the integration layer is, in our opinion, very basic, although user friendly.

All considered, those systems are more complex to learn because they have rich and elaborate APIs [12]. Simulations are software projects typically split in more files and require quite a lot of boilerplate code, since they are written in Java (or Objective-C). We feel that although such complexity is needed to cope with the general case of agent-based simulations, restricting the domain greatly helps in keeping things simple and allows for simpler tools. Moreover, the integration with different libraries, when not already provided, may be not as easy for people without a strong programming background, especially because correctly hooking into a complex framework is not particularly easy task.
III. PyNetSYM runtime model

A sound DSL design methodology advises to create a semantic model, that is to say an in-memory object model of the same subject that the DSL describes [24]. When the host language for an internal DSL (or the implementation language for an external one) is an object oriented language, the semantic model is essentially the library or framework that the DSL manipulates.

In our case, however, instead of pure objects, we have active objects with their own thread of control and the semantic model is essentially the runtime model of the agent-oriented application that executes the simulation. All the communication occurs through message passing; the default behavior of these objects is waiting for messages to arrive and then processing the incoming message with the appropriate handlers. However, full agent behavior (autonomous, goal-oriented, pro-active) can be implemented simply by overriding the main run-loop in a subclass.

The main entry point for a simulation is the Simulation class, that is one of the few domain classes not inheriting from Agent. Its main responsibilities are: (i) to define the simulation parameters (e.g., the number of simulation steps or the size of the network); (ii) to build a command line option parser that can take the parameter values from the command line; (iii) to define which objects should cover determined crucial roles for the simulation, such as the Configurator object; (iv) to create the Address-Book, that resolves the agent addresses to the actual agents, the Node-Manager and other simulation specific infrastructural agents; (v) to actually run the simulation.

The Node-Manager is responsible for (i) creating the new agents, passing them the appropriate initialization parameters and (ii) monitoring them, so that their termination (exceptional or not) is managed. Typically, in huge simulations we do not want the idle agents to be in RAM and we shut them down after a while. The Node-Manager tracks their non-exceptional termination and is able to bring them back should they receive a wake-up message.

The Configurator sends the Node-Manager the appropriate messages in order to do the initial setup for the simulation. Different Configurator objects can be chosen when defining the simulation. For example, we provide the HomogConfigurator (short for homogenous configurator) that takes the network size \( n \), a node class and the nodes initialization parameters as input parameters and has the Node-Manager create \( n \) nodes of the specified type. Another configurator is the FileLoaderConfigurator that reads a network specification from a file (e.g., a graphml file) and sets the network up accordingly.

In order to find social network simulations to test our software, in [33] and [34]: (i) we analyzed many probabilistic social network generation models (including Watts-Strogatz (WS), [35], Barabási-Albert (BA) [36], Transitive Linking (TL) [37] and the Biased Preferential Attachment (BPA) [38]) (ii) we singled out a general structure that is shared by most models and (iii) we based the semantic model on that structure. Eventually, we applied it successfully to non-generative simulations, such as disease infection over networks simulations.

The general structure of the simulation is presented in Fig. 1. Essentially, in most network generation models, at each step some nodes are created, destroyed and/or are subject of some actions. However, each model differs in the criteria used to choose the nodes: (i) to be activated, (ii) to be destroyed and (iii) to be created. When a node is activated, it performs some actions according to its behavior. Notice that the general structure does not change significantly even when introducing some agent-ness in the simulations, e.g., introducing goal-directed behavior, until we do not account for pro-active agents.

Essentially, no matter the sophistication of the simulated model, the meta-model can be instantiated as an actual generation model simply providing the specifications of: (i) the selection process of the groups of nodes to create, activate or destroy, which is performed by the Activator agent; (ii) the behavior of the nodes themselves.

Our choice of test-bed examples, i.e. “agent-oriented variants” of classic probabilistic social network generation models, biased our toolkit towards Discrete Event Simulation (DES). In fact, in such models, it is sufficient that agents are purely reactive. However, many simulations have similar requirements and not necessarily need full agent-ness, and even when the actual agent’s behavior becomes more complex and “intelligent”, we can use our software without modifications. In fact, our concurrency model does not require the rigid structure of DESs and the agents may be defined with pro-active behavior (e.g., with the goal to increase their social network).

Since the models we described are essentially DESs, in Fig. 1 we also show a Clock, which sends Tick messages to interested agents (in our case, only the Activator), so that ages in the simulations are clearly divided.

When the activator has selected the nodes to be activated, it sends them an Activate message that triggers the following course of actions. The more frequent action is to select another agent and send it a message to: (i) create a link towards the node; (ii) destroy a link with the agent. Some models also account for the refusal of the receiving agent to create or destroy the connections.

IV. Defining a DSL for Network Simulations

In Subsection II-B we mentioned the advantages of a DSL in terms of ease of development for both programmers and non-programmers because of the increased expressivity. In Section III the semantic model underlying the DSL has been described. In this Section we show the general structure of our DSL. We also present an example using a pseudo-language; although the syntax is slightly modified in order to improve typesetting in the two-column layout, its semantics is basically that of a Python program, the actual host language we used.
Our choice was motivated by: (i) Python focus in readability; (ii) ease of use; (iii) availability of useful libraries (numpy [39], scipy [40], matplotlib [41], networkx [31], igraph [22]) and consequently (iv) widespread adoption in many scientific areas; (v) solid choice of concurrency frameworks (e.g., gevent [42], stackless python [43]); (vi) powerful REPL implementations (built-in or ipython [44]). However, from our point of view, the host language is almost an implementation detail, since other object oriented high level languages such as Ruby or Groovy could have provided essentially the same features.

Although the pseudo-language is mostly self-explanatory, here we review some aspects that may nonetheless be surprising, especially for readers with experience in static languages such as Java or C++:

(a) a class definition class A(B) means that class A is a subclass of class B
(b) methods and handlers must have an explicitly specified formal parameter conventionally called “self” that refers to the object the method (or the handler) is invoked on.
(c) methods can be invoked with named parameters as well (this makes the method call more understandable without having to provide the method signature — that in our case is typically defined in the super-classes)
(d) sets are first class citizens and they behave as mathematical sets (also notation-wise)
(e) there are hash-maps and hash-map literals can be defined with the syntax: {key : "value", ...}
(f) everything defined directly in the class body becomes a class attribute; methods are also class attributes (however, they are callable attributes and essentially behave just like regular methods of other OO languages)
(g) instance attributes are usually simply defined in the constructor and we omit them in the example for the sake of brevity of presentation
(h) when a method is invoked on an instance object, that object is passed as actual argument as the “self” formal parameter of the method: e.g., graph.neighbors() invokes the neighbors method defined in graph’s class and passes the graph instance as its self argument

(i) classes are objects in they own right; they are callable objects and behave as factories, i.e., they return an object instance, when they are called
(j) classes can be defined inside other classes

As a consequence of properties (f), (i) and (j), an inner class can be invoked as a method of the enclosing class and returns the appropriate instance object.

Our simulations have two distinct logical elements:
1) the essentially imperative/object oriented description of the agents behavior (e.g., the nodes and the ACTIVATOR)
2) the mostly declarative description of the simulation structure: (i) presence/absence of an ACTIVATOR; (ii) choice of the CONFIGURATOR and its options; and (iii) simulation options specification

In order to specify the behavior of the agents we essentially use the host language directly. Agents are represented as regular objects inheriting from some AGENT subclass so that they can send and receive asynchronous messages. AGENT instances have an “id” attribute representing the address and a send method, that can be invoked to communicate with other agents, providing their id. Additional named actual parameters that are passed to the send method are forwarded to the appropriate handler of the receiving agent. In AGENT subclasses, we also define message handlers that are invoked when the corresponding message is received. Message handlers are inherited and can be overridden. Although in most simulations the actors are purely reactive, it is possible to modify their behavior to make them pro-active overriding their run-loop.

The other logical element defining a model is the SIMULATION class. A SIMULATION is not an agent and is essentially the executable specification of the simulation that collates together all the other elements. A SIMULATION object has a run method that is called to execute the simulation. When run is called, command line parameters are taken into account, but are overridden by named parameters directly passed to the run method, such as as shown in Fig. 2.

One of the most important elements in the SIMULATION definition is the specification of the simulation options. Since
writing command line argument parser is boring and time consuming, our system automatically turns the definition of the simulation options in the actual parser, which is eventually used to process the parameters from the command line.

Moreover, we use some meta-programming to interpret some class attributes (e.g., “options” in ACTIVATOR or “node_options” in HOMOGCONFIGURATOR) to have the specified options passed to the objects constructors at construction time. In general, we use meta-programming and other regular language features (e.g., abstract base classes) to make sure that the simulation fails as early as possible if not correctly specified. For example, if some necessary field or option is missing, it fails either at “compile time” (when the classes are evaluated) or when the objects are instantiated, but before the actual process starts.

In Fig. 3 we show the code for a simulation implementing the BA model. Line numbers reported in the following paragraphs refer to Fig. 3. The BA model starts with $n_0$ nodes and no edges. At each step a new node with $m$ random links is added. The $m$ links are directed towards nodes with a probability proportional to their degree, this strategy is called preferential attachment.

In Lines 25–34 the simulation configuration is specified. Line 26 specifies that the BA-ACTIVATOR (defined in lines 14–24) is the activator to be used. Since classes are first class citizens, they can be stored in variables like any other object.

Lines 27–29 specify the command line options, their default values and the type of the parameters. When values for some options are not specified neither when calling run, nor in the command line, the default value is used. The type is only used to parse values from the command line.

The CONFIGURATOR object (Lines 30–33) is responsible for creating the starting configuration for the simulation: in this case, it is a subclass of HOMOGCONFIGURATOR, so all the starting nodes: (i) have the same type, BA-NODE (line 31), and (ii) are passed option values specified in the “node_options” variable (line 32). This is an example of what we already mentioned: we declaratively specify that nodes created at the CONFIGURATOR’s request should be passed the “starting_edges” option value. In the HOMOGCONFIGURATOR definition we specify that it also reads the “network_size” property to choose the initial number of nodes to create: essentially CONFIGURATOR inherits the value of its options attribute from HOMOGCONFIGURATOR.

In Lines 1–13 and 14–24 there are the specifications for the nodes and the activator respectively. The nodes are specified providing a handler for the an Activate message. Moreover, in the NODE class we defined handlers for the link_to message, so that the graph is updated with the information.

The ACTIVATOR accepts both a Tick message (from the clock) and the Created-Node message. The latter is the NODEMANAGER’s answer to the message the ACTIVATOR sent on line 16, signaling that the required node has been created and is ready to be activated. Moreover, the ACTIVATOR class has an attribute options, that is used by the SIMULATION to choose which of the simulation options need to be passed to the ACTIVATOR’s constructor. In Line 16 the ACTIVATOR passes to the NODEMANAGER: (i) the class of the agent to create in the “cls” parameter and (ii) the arguments to pass to their constructor with the “parameters” parameter.
V. PyNetSYM Concurrency Model

In Section III we defined the runtime model of our simulation framework. Essentially, we assumed that agents do have their own thread of control, but did not specify how concurrency was implemented. The traditional approach in agent-based frameworks is to implement concurrency mainly with threads and to provide cooperative multitasking as an option [45].

On the other hand, we decided to use gevent [42], a networking and concurrency framework that uses coroutines to implement cooperative tasks, called in gevent lingo “greenlets”. Coroutines are a generalization of subroutines allowing multiple entry points for suspending and resuming execution at certain locations [46]. Gevent also provides the greenlets scheduler. Frameworks such as gevent are popular for writing programs with very high concurrency, since greenlets: (i) are less expensive to create and to destroy; and (ii) do not use memory structures in kernel space. In fact, greenlets live entirely in the user-space, thus context switches between different greenlets are inexpensive as well.

This is particularly relevant in our case since in ABM it is required to potentially support millions of concurrent agents, and, since greenlets use no kernel resources, their number is limited only by the amount of available RAM. On the other hand, experience suggests that a modern operating system can roughly support few thousand threads.

In PyNetSYM an agent is essentially a greenlet with a mailbox where it receives messages (implemented with a gevent queue). However, since gevent also provides cooperative networking primitives, agents can transparently communicate with agents on remote machines. Moreover, PyNetSYM promotes a programming model where idle agents do not necessarily store state in-memory, and consequently they can be destroyed and recreated only when needed, thus using even less RAM.

In Table I we report execution times of a simple benchmark performed with a different number of threads/greenlets. Essentially, for this simple benchmark, greenlets are roughly ten times faster than threads. In the benchmark program, we implemented reactive agents both with threads and greenlets. The main agent spawns a new agent and sends it a message with the numeric value of the number of agents to spawn, then it waits for a message on its own queue. When receives a message with value \( n \), each secondary agent: (i) spawns a new agent and sends it a message with the value \( n-1 \) if \( n \neq 0 \); (ii) on the other hand, if \( n = 0 \), it means that all the agents have been created and sends a message to the main thread. Notice that the agents do not exist all at the same time: once they have received and sent a message, they terminate. Further comparisons between greenlet and thread performances can be found in [47].

VI. Conclusion

In this paper, we presented PyNetSYM, a novel language and runtime for network specific simulations. PyNetSYM is built for the generative approach to science typical of Agent-Based Modeling (ABM). We advocate there is a strong need for tools that are both easy to use (especially for people with little programming background) and able to tackle large scale simulations, using remote high-performance machines and potentially distributing the computation on multiple servers. Therefore, while maintaining our toolkit simple and easy to use is our primary goal, efficiency is our second priority, since nowadays the frontier is research on networks of huge size.

Specifically, we created PyNetSYM: (i) to easily support small and huge networks (using either in-memory and on-disk network representations) and (ii) to be as easy to use on personal machines or on remote servers, even for researchers without strong programming background.

We designed PyNetSYM so that all the entities, both the infrastructural ones and those under simulation, are agents: defining the network simulation is essentially a matter of defining (i) the behavior of the nodes and (ii) a few additional simulation parameters (e.g., storage strategy and user-customizable options).

Given the merits of Domain-Specific Languages (DSLs) in general, and specifically those concerning how to make development easier both for programmers and non programmers alike, we decided to create a DSL to drive PyNetSYM simulations, so that it is possible to write programs that are machine-executable high-level formulations of the problem to solve. Specifically, our DSL is an internal DSL over Python. As PyNetSYM provides the simulation engine, the simulation can be written in a simple file using our DSL. Thus, PyNetSYM models are very easy to deploy (copying around a single file is sufficient) and can also be effortlessly shared between researchers. Moreover, PyNetSYM models can also be written and executed interactively from a Read-Eval-Print Loop (REPL).

We also implemented some classic network generation models with PyNetSYM and found that the resulting programs are readable and fluent; in fact, they very similar to a pseudo-code formulation of the models, even though they are efficiently executable.

Our results show that our approach is successful in providing a friendly and easy to use environment to perform agent-based simulations over social networks. Agent-based simulation is a powerful conceptual modeling tool for social network simulations and with the present work we contribute a natural and expressive way to specify social network simulations using a DSL.

<table>
<thead>
<tr>
<th># items</th>
<th>Thread execution (s)</th>
<th>Greenlet execution (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.022</td>
<td>0.002</td>
</tr>
<tr>
<td>500</td>
<td>0.110</td>
<td>0.009</td>
</tr>
<tr>
<td>1000</td>
<td>0.224</td>
<td>0.018</td>
</tr>
<tr>
<td>10000</td>
<td>2.168</td>
<td>0.180</td>
</tr>
<tr>
<td>100000</td>
<td>21.85</td>
<td>1.796</td>
</tr>
<tr>
<td>1000000</td>
<td>223.3</td>
<td>18.24</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENT

Stimulating discussions with Prof. A. Poggi are gratefully acknowledged.

REFERENCES


